Automated Theorem Proving for Modal Logic using a SAT solver

Ethan Nguyen\textsuperscript{1}[u6672582], Supervisor: Rajeev Goré\textsuperscript{1}

Australian National University, Acton ACT 2601, AUS
\{ethan.nguyen, rajeev.gore\}@anu.edu.au

Abstract. Theorem Proving for Modal Logic has traditionally been done with semantic tableaux methods. We are nearing the limit for speed when it comes to these methods. For classical propositional logic, theorem provers using SAT solvers are very efficient. We introduce a method for Modal Logic which incorporates a SAT solver for a speed increase. As a code base we use a state of the art theorem prover for intuitionistic logic, IntSAT, by Claessen and Rosen. We yield good preliminary results and see that there is a lot of room for improvement.

Keywords: Modal Logic · Theorem Proving · SAT Solver.

1 Foreword

As this is a report, and not a published research paper, I will treat it like so. This is still a formal piece of writing, but I will give insights into the difficulties encountered and the ways they influenced the method and the final result.

2 Introduction

Modal Logic is an extension of propositional logic, which includes modalities, there are notions of time, obligation, permission, necessity, possibility and more. This is important because a lot of how we describe intelligent behaviour has to do with modalities. For artificial intelligence (AI), an agent would have to understand when an event is happening, what it is allowed to do, what is possible and more [9]. Not only this, but the agent would have to reason with this modality riddled statement, that is, to determine the truth hood of the statement. This is theorem proving for modal logic, also known as reasoning for modal logic. However, reasoning for modal logic is reaching the limits of efficiency, we need reasoning in modal logic to be efficient as every second counts in real time systems, where Modal Logic has a lot of applications in, for example AI. Modal Logic also has applications in game theory, mathematics, temporal reasoning, philosophy and linguistics [2].

Automated Theorem Proving is the act of using the computer to determine whether a certain logic formula is valid. For classical propositional logic, validity
is defined as: for every possible assignment of the Boolean literals which make up the formula, the truth value of the formula is true. A Boolean literal is value which represents true or false. An example of validity is $p \lor \neg p$, where $p$ is a Boolean literal:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\neg p$</th>
<th>$p \lor \neg p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

We can see from the truth table, that every possible assignment of the Boolean literal $p$ makes the formula $p \lor \neg p$ true. Thus, $p \lor \neg p$ is valid.

This stems from satisfiability, a logic formula is said to be satisfiable if there exists an assignment of the Boolean literals in the formula which makes the formula true.

A simple way to determine validity from satisfiability for an arbitrary formula $A$, is to check if $\neg A$ is satisfiable. If $\neg A$ is satisfiable, then $A$ is not valid as there exists an assignment of Boolean literals which make $\neg A$ true which is the same as $A$ being false. If $\neg A$ is not satisfiable, then $A$ is valid, as there does not exist an assignment of Boolean literals which makes $\neg A$ true, in other words there is no assignment which makes $A$ false.

A method of determining satisfiability is via semantic tableaux. Typically, theorem provers for modal logic use this method. Another method for determining satisfiability is via Satisfiability solvers (SAT solvers). Recently, research into SAT solvers have gained a lot of traction, due to a lot of optimisation and algorithmic techniques SAT solvers are efficient and fast. However, most SAT solvers are only designed for classical logic.

Claessen and Rosen introduce a new method for theorem proving in intuitionistic logic using a SAT solver [3]. Intuitionistic logic is a non-classical logic akin to modal logic. Intuitionistic logic is propositional logic without the law of excluded middle. This is important as SAT Solvers are typically designed for propositional logic. Claessen and Rosen’s method is state of the art for intuitionistic logic. Not only that, intuitionistic logic has an underlying Kripke model which is similar to modal logic. Due to this, we build off Claessen and Rosen’s work by using their code as a base and understanding their procedure for proving in intuitionistic logic.

Claessen and Rosen mention in their future work section generalising IntSAT to work on modal logic. The key insight is to manipulate a modal into 3 types of modal clause forms. A clause is an expression of a finite amount of Boolean
Automated Theorem Proving for Modal Logic using a SAT solver

3 Related Work

We are theorem proving for modal logic, we compare our work against state of the art: BDDTab[6], FaCT++[11], InKreSAT[7] and *SAT[8]. BDDTab replaces standard DPLL-based approaches to modal satisfiability with BDDs. FaCT++ is a reasoner for description logics and uses new architecture with a wide range of optimisations including several novel ones. InKreSAT is a prover for modal logic, it also works with a SAT solver, the difference is the way it is used. InKreSAT reduces a modal satisfiability problem into a boolean satisfiability problem and then solves it with a SAT solver, contrasted with how we consider the underlying kripke model and apply the SAT solver to each possible world. Lastly, *SAT also uses SAT solvers, it is built on top of a SAT solver and investigates the applicability of a SAT solver to modal and descriptive logics.

Other related work includes Salhi’s work, On Satisfiability Problem in Modal Logic S5[10], which explores different methods of finding satisfiability in Modal Logic S5, with plans to create a DPLL like algorithm for deciding satisfiability in Modal Logic S5.

4 Modal Logic

This section is paraphrased from Modal Logic for Artificial Intelligence by Rosja Mastop[9].

4.1 Definition

Modal Logic is developed for reasoning about modalities. Modalities are notions of time, necessity, possibility, belief, knowledge and much more. In this paper we deal with classical normal modal logic, which is also known as K modal logic. K modal logic is concerned with reasoning about possibility and necessity.

Necessity is expressed through the □ unary operator, possibility is expressed through the ◊ unary operator. □φ reads it is necessary that φ. ◊φ reads it
is possible that \( \varphi \). We add these unary operators on top of propositional logic language, meaning we still have the operators used in propositional logic.

\[
L_{modal} := \ p \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \varphi \to \varphi \mid \varphi \leftrightarrow \varphi \mid \neg \varphi \mid \Box \varphi \mid \Diamond \varphi \mid \bot
\]

### 4.2 Kripke Model

Analogous to how propositional logic is evaluated in truth tables, modal logic is evaluated through kripke models. A kripke model consists of worlds, accessibility relations and valuations.

**Definition 1.** A Kripke Model is a tuple \( M = \langle W, R, V \rangle \), such that

- \( W \) is a non-empty set of possible worlds
- \( R \subseteq (W \times W) \) is a binary accessibility relation.
- \( V : W \to \mathcal{P}(\text{var}) \) is the set of valuations at a given world for a set of propositions \( \text{var} \). Note \( \mathcal{P}(\text{var}) \) is the power set of \( \text{var} \). A proposition \( p \) is true in world \( w \) if \( p \in V(w) \), if \( p \notin V(w) \) then \( p \) is false in world \( w \).

Then, given a model \( M = \langle W, R, V \rangle \):

1. the formula \( \Box \varphi \) is true in a possible world, if and only if, \( \varphi \) is true in every possible world that is accessible from \( w \).
2. the formula \( \Diamond \varphi \) is true in a possible world, if and only if, \( \varphi \) is true in some possible world that is accessible from \( w \).

### 5 Modal Clauses

Claessen and Rosen mention in their future work that generalising INTSat to modal logic, that in order to prove a modal formula \( A \) they generate a fresh literal \( q \) and represent it as \( \Box(A \to q) \). Then they consider the constraints of the three following shapes:

1. \( \Box p \), for a propositional logic formula \( p \)
2. \( \Box(a \to \Box b) \), for propositional logic literals \( a \) and \( b \)
3. \( \Box(a \to \Diamond b) \), for propositional logic literals \( a \) and \( b \)

Goré and Nguyen had similar ideas as Claessen and Rosen. Goré and Nguyen consider 5 modal clause forms, however, these 5 forms could actually be simplified to 3. Building on this idea from Claessen and Rosen, in this paper for a modal logic formula \( A \) we manipulate into the following 3 similar Modal Clause (MC) forms:

1. MC1: a propositional logic formula \( p \)
2. MC2: propositional logic literal implies box propositional logic literal \( (a \to \Box b) \), for propositional logic literals \( a \) and \( b \)
3. MC3: propositional logic literal implies diamond propositional logic literal \( (c \to \Diamond d) \), for propositional logic literals \( c \) and \( d \)

That is, \( A \equiv \bigwedge MC1 \land \bigwedge MC2 \land \bigwedge MC3 \). Once we manipulate a formula \( A \) into this form we can use it in the procedure. My supervisor Rajeev Goré provided me with the algorithm to manipulate any modal formula \( A \) into the 3 modal clause forms \( MC1, MC2 \) and \( MC3 \).
6 Procedure

6.1 Using satisfiability to get validity

If we wanted to ascertain if a formula $A_1$ was valid using satisfiability, we check if $\neg A_1$ is satisfiable or not. If it is not satisfiable then the formula $A_1$ is valid as we cannot find a satisfying assignment which makes $\neg A_1$ true, in other words we cannot find a satisfying assignment which makes $A_1$ false. If $\neg A_1$ is satisfiable then $A_1$ is not valid.

6.2 Preliminary adjustments

Given a modal formula $A_1$ which we are determining validity for, similar to Claessen and Rosen, we manipulate the formula into $(A_1 \rightarrow q)$ for a fresh literal $q$ ($q$ does not occur anywhere in $A_1$). We convert $A_1 \rightarrow q$ into negation normal form (NNF), so the formula is now $\neg A_1 \lor q$, where $\neg A_1$ is also in NNF. Notice, $A_1$ is negated, this is how we will be using satisfiability to get validity.

We then manipulate $\neg A_1 \lor q$ into the 3 modal clause forms, $\neg A_1 \lor q \equiv \bigwedge MC1 \land \bigwedge MC2 \land \bigwedge MC3$. We then input this into our algorithm, ModalSAT. ModalSAT will return the atoms which were conflicting if the formula that was inputted in conjunction with the goal is unsatisfiable, otherwise it will return nothing.

6.3 SAT Solver

Like Claessen and Rosen, we use a standard (classical) SAT-Solver $sat$, that can do the following operations:

1. procedure newSolver()
2. procedure addClause(sat, c)
3. procedure satSolve(sat, A, q)

Exactly the same as Claessen and Rosen, with wording change from $satProve$ to $satSolve$ as the SATSolver is not proving (searching for validity) but rather trying to find a satisfying assignment.

The procedure newSolver creates a new, unconstrained SATSolver. The procedure addClause takes a SATSolver sat and adds a clause c, and adds the clause r as a constraint to sat. Note, the clause added is in NNF.

The procedure satSolve(sat, A, q) tries to find a satisfying assignment of the clauses already added to the SATSolver sat, the set of assumptions $A$ and the goal $q$. Thus satSolve will attempt to find a satisfying assignment for $\bigwedge C \land (A \rightarrow q)$ where $C$ is the set of clauses added as a constraint to the instance sat. satSolve can return the following:

1. Satisfiable($M$), this means that there is a satisfying assignment, $M$ is the model which contains the assignments of atoms which satisfy the clauses added to sat, the set of assumptions $A$ and $q$ the goal.
2. Unsatisfiable($A'$), this means that no satisfying assignment could be found. $A'$ is the assignment of the atoms which conflict with each other which is causing the SAT Solver to be unable to find a satisfying assignment.
6.4 Top level prove algorithm

Algorithm 1 prove(A)

\[
\begin{align*}
\text{fml} & \leftarrow (A \rightarrow q) \quad \text{for some fresh literal } q \\
\text{fml} & \leftarrow (\neg A \lor q) \quad \text{where } A \text{ is in NNF} \\
\text{fml} & \leftarrow (\neg A \lor q) \equiv \land MC1 \land MC2 \land MC3 \\
\text{sat} & \leftarrow \text{newSolver()} \\
\text{for } p \in MC1 & \text{ do} \\
& \quad \text{addClause(sat, } p) \\
\text{end for} \\
\text{mass} & \leftarrow \text{ModalSAT(sat, } q, MC2, MC3) \\
\text{if } mass == \text{Satisfiable} & \text{ then} \\
& \quad \text{return } \text{Countersatisfiable} \\
\text{else} & \text{ return } \text{Valid} \\
\text{end if}
\end{align*}
\]

6.5 ModalSAT algorithm

Different from Claessen and Rosen, we make our algorithm recursive. This is inspired from Fiorentini, Goré and Graham-Lengrand’s paper A Proof-Theoretic Perspective on SMT-Solving for Intuitionistic Propositional Logic [4]. They present a recursive version of IntSAT. We make a recursive ModalSAT algorithm. My supervisor, Rajeev Goré, provided me the pseudo-code outlining the ModalSAT algorithm.

Note, for line 16, the reason why we learned \( \text{ActivatedAs} \rightarrow \neg c \), is because of the following. Line 7, the contrapositive of \( c \rightarrow \Diamond d \) is \( \Box \neg d \rightarrow \neg c \). Since \( \tau_1 \) is not \textit{Satisfiable} then we know \( B \rightarrow \Box \neg d \). Thus by transitivity, \( B \rightarrow \neg c \). Since \( \text{ActivatedAs} \rightarrow B \) then by transitivity again we know that \( \text{ActivatedAs} \rightarrow \neg c \).

7 Results

We test our procedure against the Logic Work Bench (LWB) benchmarks [1]. This set of benchmarks are widely used for modal reasoning. LWB benchmarks include multiple set of benchmarks for different modal logics, we only use the benchmarks for K Modal logic as we can only support this at the moment. In the LWB K modal logic benchmarks they test different properties of K modal logic, an example is \( ph \), which tests pigeonhole formulas. For each property tested, there is a set of formulas which are valid and another which are not valid, indicated by a \( p \) (provable, which means valid) or \( n \) (not provable, which means invalid) file name. An example of the file name for a set of formulas is \( k\_ph\_p \),
Algorithm 2 ModalSAT(sat, q, MC2, MC3)

1: \( \tau_0 \leftarrow \text{satSolve}(\text{sat}, \neg q) \)
2: if \( \tau_0 == \text{Unsatisfiable}(A') \) then
3: \( \text{return } A' \) \{we can exit the function since it is unsatisfiable\}
4: else if \( \tau_0 == \text{Satisfiable}(M) \) then
5: \( B \leftarrow [b \mid (a \rightarrow \Box b) \in MC2 \text{ and } a \in M] \) \{Gather all boxes which are activated\}
6: \( AB \leftarrow [(a \rightarrow \Box b) \mid (a \rightarrow \Box b) \in MC2 \text{ and } a \in M] \)
7: for \( \iota \leftarrow (c \rightarrow \Diamond d) \in MC3 \text{ and } c \in M \) do
8: \( \text{sat}_1 \leftarrow \text{newSolver}() \) \{Prepare a jump to a successor world, create a new SAT Solver\}
9: for \( b \in B \) do
10: \( \text{addClause}(\text{sat}_1, b) \) \{In this successor world, all activated \( b \) such that \([b] b \) was activated are true\}
11: end for
12: \( \tau_1 \leftarrow \text{ModalSAT}(\text{sat}_1, \neg d, MC2, MC3) \) \{does \( (B \rightarrow \neg d) \) hold\}
13: if \( \tau_1 \neq \text{Satisfiable} \) then
14: \( \text{ActivatedAs} \leftarrow [a \mid (a \rightarrow \Box b) \in AB \text{ and } b \in \tau_1] \) \{all a’s which were activated which drove each \( b \in B \}\}
15: for \( c' \rightarrow \Diamond d' \in MC3 \text{ and } d' == d \) do
16: \( \text{addClause}(\text{sat}, \text{ActivatedAs} \rightarrow \neg c') \) \{Learned that \( \text{ActivatedAs} \rightarrow \neg c' \)\}
17: end for
18: \( \tau_2 \leftarrow \text{ModalSAT}(\text{sat}, q, MC2, MC3) \) \{Try the root world again with new learned clauses\}
19: if \( \tau_2 \neq \text{Satisfiable} \) then
20: \( \text{return } \text{used} \leftarrow \text{all true literals in sat} \) \{Not satisfiable, return all true literals in sat\}
21: end if
22: end if
23: end for
24: \( \text{return } \text{Satisfiable} \)
25: end if
this means testing pigeonhole formulas which are valid for K modal logic.
In each property that is tested, there are multiple formulas to a max of 21
formulas in a property. The idea is, each formula is bigger than the last. The
typical way to use these benchmarks, is given a certain timeout, how many
formulas can the prover determine validity for, where each formula must run for
less than the timeout value. In our tests we use a 30s time out. For example,
in the results table a value of 10 would indicate that the reasoner solved every
formula in that file from 1 - 10 in less than 30s for each formula. The best possible
score 21, as that is the amount of formulas there are in a file.

<table>
<thead>
<tr>
<th>subclass</th>
<th>BDDTab</th>
<th>FaCT++</th>
<th>InKreSAT</th>
<th>*SAT</th>
<th>ModalSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch_n</td>
<td>18</td>
<td>10</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>branch_p</td>
<td>21</td>
<td>10</td>
<td>16</td>
<td>18</td>
<td>14</td>
</tr>
<tr>
<td>lin_n</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>path_n</td>
<td>21</td>
<td>21</td>
<td>10</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>path_p</td>
<td>21</td>
<td>21</td>
<td>10</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>ph_n</td>
<td>10</td>
<td>13</td>
<td>21</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>ph_p</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>poly_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>t4p_p</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>5</td>
</tr>
</tbody>
</table>

ModalSAT was ran on a Ryzen3600X@3.8GHz with 16GB memory. Due to
technical difficulties I was unable to get the other reasoners running on the same
computer. Thus I used the results table presented in Implementing Tableau Cal-
culi Using BDDs: BDDTab System Description by Goré, Olesen and Thomson
[6]. Thus the BDDTab, FaCT++, InKreSAT and *SAT tests were ran on an In-
tel 3.4GHz CPU with 8GB of memory. I do realise the computer can play a role
in the speed of the provers, I endeavour to get this working in the near future.
For now, please take the results at face value.

We can see that at the moment we do not match the state of the art for
a lot of the test cases, however we have great results on ph_p. These are good
preliminary results, we believe there is a lot that can be done to optimise the
code. There is a lot of potential for our approach.

8 Future Work

ModalSAT was built in Haskell, on top of IntSAT. In the current implement-
tation, there are a few repeating functions, some for debugging purposes and
some require refactoring. After refactoring and removing debugging functions
we should see an increase in performance. Not only that, but the current im-
plementation is naive, the use of data structures have room for improvement.
For example, lists could be replaced with hash tables. Furthermore, there is
a lot that can be done in parallel, an example is getting all true variables in
the current instance of the SAT solver, or running the SAT solver on successor worlds.

We look to extend the current implementation to be able to work on other modal logics including, T, K4, and S4.

9 Conclusion

In this paper we introduced a method for determining the validity of K modal logic formulas using a SAT Solver, this was done by manipulating the negation of a modal formula A into the 3 Modal Clause forms. From here we check for satisfiability using a SAT Solver at each world in the underlying kripke model. Using LWB benchmarks, our algorithm shows good preliminary results, even beating the state of the art for one problem set. There is a lot to be optimised, including the data structures used.

In the future, we will extend this method to deal with other modal logics including, T, K4, and S4.

10 Difficulties

10.1 Choice of Language

We used Haskell in this project, but we had the choice between Prolog or Haskell. In Prolog we would not be able to use well optimised SAT Solvers like MiniSAT, in Prolog these SAT Solvers did not return the atoms used in finding a satisfying assignment or an unsatisfying assignment which was crucial to the algorithm. If I were to use Prolog I would be using a basic implementation of a SAT Solver written by Rajeev Goré. The cons of this was the speed of the SAT Solver, however, the pros were that Rajeev Goré knows Prolog extensively and could assist me when I would get stuck writing Prolog code. Writing it in Prolog would suffice to test that the algorithm works and I would be able to gain the same knowledge out of the project. If we were to choose to write in Prolog, we would not be working off IntSAT.

Haskell would be more difficult to implement since Rajeev Goré would not be able to assist in the actual writing of Haskell code. However, Rajeev Goré could still assist me in the logic of the implementation and debugging. The pros of using Haskell are we are building off the state of the art of theorem proving for intuitionistic Logic and we are able to use state of the art SAT Solvers like MiniSAT. In the end we ended up choosing Haskell, which had a few more problems than expected.

10.2 Working off IntSAT

Now that we chose to do in Haskell we were working off IntSAT, this itself was a challenge. The way they implemented their code was very obtuse, at times
it wasn’t very intuitive. We consulted with Dr Ranald Clouston the lecturer for COMP1100 Haskell, he commented that the way Claessen and Rosen implemented the code was very imperative and was not necessary. The documentation for IntSAT was lacking as well, thus it took a lot of time to first understand their code before we tried to modify it.

10.3 Monads in Haskell

A great difficulty we encountered was Claessen and Rosen’s use of Monads in Haskell. Most of the Monad usage seemed unnecessary and neither Rajeev Goré and I were able to uncover what it really is and why they used it. We consulted with Dr Jeremy Dawson, who is experienced in functional programming, regarding the use of Monads:

Hi Raj,
I’m very confused.
First, in Clausify.hs, it says import qualified Data.Map as M I think this means it imports a data structure called Data.Map and calls it M. Thus later on in that file you get M.Map (a type constructor) and M.empty, M.lookup etc


But also in Clausify.hs we have M used as a type constructor, eg namesAnd :: Form → M [Name] which is unfortunately redefined (I think) newtype M a = M (Cache → Int → (a, Cache, Int, Seq Clause, Seq ImplClause)) and shown to be a Functor and a Monad, etc instance Functor M where instance Monad M where

I’d say these two uses of M are independent, except that, unfortunately, try as I might (I’ve spent ages on this) I can’t see where it finds any definition for the type constructor M. So it might be best to ask Ranald or one of his tutors who uses Haskell routinely. Unfortunately Haskell (unlike Coq!) doesn’t have a feature for you to ask where some word is defined.

CHeers,
Jeremy

References

Automated Theorem Proving for Modal Logic using a SAT solver


